Univerza v Ljubljani
Fakulteta za računalništvo in informatiko
Fakulteta za matematiko in fiziko

David Kotar

Študija prehoda na arhitekturo ARM Cortex-M v sistemu za avtomatizacijo hiše

diplomsko delo
univerzitetni študijski program prve stopnje
računalništvo in matematika

Mentor: izr. prof. dr. Patricio Bulić

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Opravite študijo prehoda iz 8-bitne arhitektire ATmega328 na arhitekturo ARM Cortex-M v primeru obstoječega sistema za avtomatizacijo hiše. Na novi platformi implementirajte del sistema in ocenite, kakšna strojna oprema bi bila potrebna za prehod na novo arhitekturo ter kakšne spremembe so potrebne v programski opremi. Za potrebe študije uporabite razvojno ploščico STM32F4 Discovery, ki vsebuje zmogljiv sistem na čipu, katerega jedro je procesor ARM Cortex M4.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani David Kotar, z vpisno številko **63080081**, sem avtor diplomskega dela z naslovom:

Študija prehoda na arhitekturo ARM Cortex-M v sistemu za avtomatizacijo hiše

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Patricia Bulića,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 15. avgusta 2014 Podpis avtorja:

Zahvaljujem se vsem, ki so mi v času pisanja diplome stali ob strani.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
TRIAC	Triode for alternating current	Trioda za izmenično napetost
RAM	Random access memory	Spomin z naključnim dostopom
OC	Output compare	Primerjalni kanal
OPM	One-pulse mode	Eno-pulzni način
CPU	Central processing unit	Centralna procesna enota
PSP	Interrupt service routine	Prekinitveno-servisni program
DSP	Digital signal processing	Digitalno procesiranje sigalov
USB	Universal serial bus	Univerzalno serijsko vodilo
PWM	Pulse width modulation	Pulzno širinska modulacija

Povzetek

Ob razvoju sistema za avtomatizacijo hiše Assys smo z razširjanjem funkcionalnosti in kompleksnosti dosegli zmogljivostno mejo mikrokrmilnika ATmega328P. V diplomski nalogi smo opisali, katere lastnosti platforme onemogočajo nadaljni razvoj in zakaj je potreben prehod na zmogljiveši mikrokrmilnik. Študijo prehoda smo izvedli na primeru razvojne ploščice STM32F4 Discovery, katera vsebuje zmogljiv mikrokrmilnik iz družine ARM Cortex-M4. Na novi platformi smo implementirali del sistema, ki skrbi za zatemnjevanje in neposredno prikazali koristi zmogljivejše strojne opreme. V študiji smo ugotovili, da je kljub višji ceni in porabi energije mikrokrmilnika ARM, ter dodatnim stroškom ob prehodu zamenjava nujna, upoštevajoč pridobljeno zmogljivost pa tudi upravičena.

Ključne besede: sistem za avtomatizacijo hiše, mikrokrmilnik, zatemnjevanje.

Abstract

When developing a system for the home automation Assys we expand the functionality and complexity of the system and with that we achieved the highest capability of microcontroller ATmega328P. In this thesis we described the properties of the platform which prevent the further development and reasons for the need for more capable microcontroller. The study was carried out on the development board STM32F4 Discovery, which contains a powerful microcontroller from the family ARM Cortex-M4. On the new platform, we implemented part of the system which is responsible for dimming and with that we directly demonstrated the benefits of more powerful hardware. In the study, we found out that the replacement of the microcontroller is necessary despite the higher price, power consumption and additional costs during the transition. If we consider the gained performance of the new platform the replacement is justified.

Keywords: home automation system, microcontroller, dimming.

Poglavje 1

Uvod

Pri razvoju takšnih ali drugačnih računalniških sistemov ali aplikacij se včasih srečamo z omejitvami strojne opreme, kot so hitrost, velikost pomnilnika... Zgodi se, da na določeni točki izkoristimo, kar nam strojna oprema ponuja, in da je za nadaljni razvoj potrebna zamenjava le-te. V diplomski nalogi smo predstavili, s katerimi omejitvami smo se srečali pri razvoju sistema za pametne hiše Assys in izvedli študijo prehoda na zmogljivejšo platformo. Z novo platformo želimo doseči hitrejše delovanje sistema, poenostaviti razvoj programske opreme ter z zmogljivostjo predvsem zagotoviti možnost dodajanja novih funkcionalnosti v prihodnosti.

Sistem trenutno deluje na osnovi 8-bitnega mikrokrmilnika ATmega328P iz družine Atmel AVR. ATmega zmore opravljati naloge v takšnem obsegu, kot je v danem trenutku implementiran, na žalost pa ne dopušča nadgradenj. Na izredno konkurenčnem področju sistemov za pametne hiše je pomembno, da se pri razvoju sledi trendom, željam uporabnikom in da se z dodajanjem inovativnih funkcij vedno nahaja korak pred konkurenco. Vse to zahteva zmogljivo strojno platformo. Študijo prehoda smo naredili na mikrokrmilniku iz družine ARM Cortex-M4 podjetja STM, ki predstavlja vrh ponudbe v razredu splošnonamenskih mikrokrmilnikov.

V Poglavju 2 smo opisali in predstavili strojno in programsko opremo, katera je predmet te diplomske naloge. Poglavje 2.5 na kratko predstavi sam sistem za avtomatizacijo hiš Assys. Prehod med platformama smo opisali v Poglavju 3. Pogledali smo, kaj nam obe platformi ponujata ter pretehtali smiselnost prehoda. V primerjavo smo vzeli tudi alternativne arhitekture. Del sistema, ki skrbi za zate-

mnjevanje smo implementirali na novem mikrokrmilniku in to prikazali v Poglavju 4. Razvito kodo smo primerjali s staro ter konkretno prikazali prednosti nove strojne opreme.

Poglavje 2

Uporabljene komponente in programska oprema

Naloga se osredotoča na dve platformi mikrokrmilnikov: Atmel ATmega328P, ki je trenutno v uporabi, ter ARM Cortex-M4 podjetja STM, na katerem bi lahko sistem temeljil v prihodnosti. Omenili bomo tudi, katero razvojno okolje smo uporabili, ter opisali sam sistem za avtomatizacijo hiš.

2.1 Atmel ATmega 328P

ATmega so 8-bitni mikrokrmilniki iz družine AVR, ki jih je leta 1996 razvilo ameriško podjetje Atmel. Vse do danes so izredno priljubljeni, saj so zelo preprosti za uporabo. Obstaja veliko različnih konfiguracij, osredotočili pa se bomo na ATmega328P v DIP-28 ohišju [1], na katerem trenutno temelji sistem Assys.

ATmega328P vsebuje naslednjo periferijo:

- 32K bajtov bliskovnega pomnilnika za ukaze
- 1K bajtov pomnilnika EEPROM
- 2K bajtov statičnega pomnilnika RAM
- dva 8-bitna ter 16-bitni števec

Slika 2.1: Atmel ATmega328P DIP-28. Vir [2].

Slika 2.2: Programator USBasp. Vir [3].

- 23 digitalnih V/I linij
- serijski komunikacijski vmesnik UART
- med drugim še serijska vmesnika I2C, SPI...

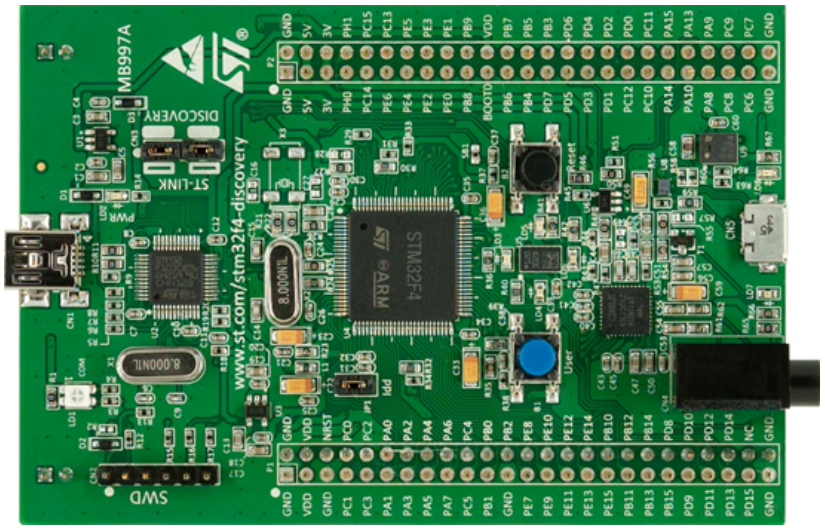
Napajalno območje je med 1.8 in 5.5 V, vgrajen je 8 MHz oscilator, z uporabo zunanjega izvora ure pa lahko teče do 20 MHz. Poraba energije je 200μA/MHz pri 1.8V.

2.2 Programator USBasp

Za programiranje mikrokrmilnika ATmega smo uporabili programator USBasp [7]. USBasp je odprtokodni programator za mikrokrmilnike družine Atmel AVR, ki deluje na vseh razširjenih opercijskih sistemih.

2.3 Razvojna ploščica STM32F4 discovery

Rešitev smo razvijali na razvojni ploščici STM32F4 discovery [8], ki vsebuje programator ST-link za mikrokrmilnike proizvajalca STM ter mikrokrmilnik STM32F407VGT6



Slika 2.3: STM32F4 discovery razvojna ploščica. Vir [10].

v LQFP100 ohišju[9]. STM32F407VGT6 je zmogljiv mikrokrmilnik z jedrom ARM Cortex-M4, ki vsebuje enoto za računanje v plavajoči vejici, ukaze za DSP ter bogat nabor V/I naprav in spomina:

- 1M bajtov bliskovnega pomnilnika
- 192K bajtov statičnega pomnilnika RAM
- skupaj 16 različno zmogljivih števcev
- 5 16-bitnih vrat za splošen V/I
- štiri serijske komunikacijske vmesnike USART
- med drugim še vmesnike I2C, SPI, ADC, DAC, DMA, CAN, generator naključnih števil, HASH procesor, USB, ethernet...

Napajalna napetost je med 1.8 in 3.3 V, deluje pa vse do frekvence 168 MHz. Poraba energije je 238μA/MHz.[1]

[1]Poraba pri izključenih perifernih napravah

2.4 Razvojno okolje in prevajalnik

Kodirali smo v razvojnem okolju Eclipse 4.3.2 (Kepler) za C/C++[4] z vtičnikoma AVR-eclipse[5] v primeru mikrokrmilnika ATmega, ter GNU ARM Eclipse [6] v primeru Cortex-M4 podjetja STM. Vtičniki nam olajšajo uvoz knjižnic in prevajanje programov.

Uporabljena prevajalnika sta bila avr-gcc 4.7.2 (GCC 4.7.2) [11] ter GCC ARM embedded 4.8.3 [12].

2.5 Sistem za avtomatizacijo hiše Assys

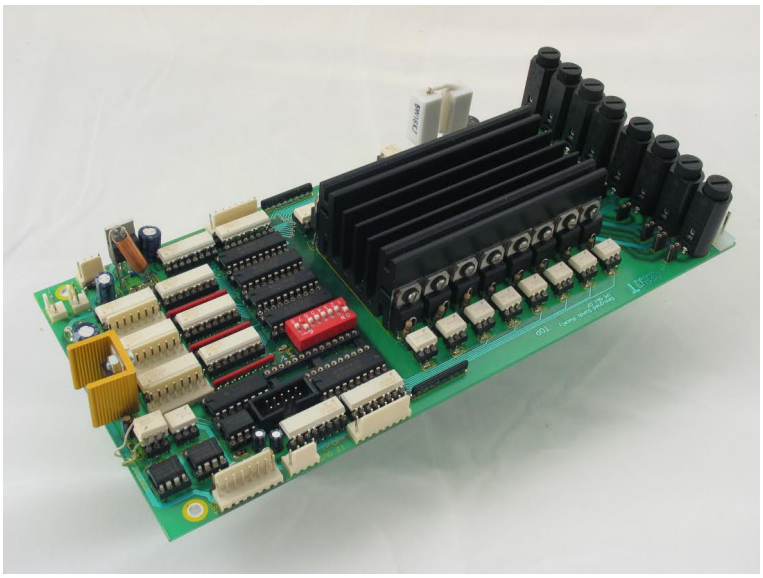
Assys[13] je sistem, ki omogoča celovito upravljanje doma, kot je avtomatsko zastiranje senčil in prilagajanje svetlosti luči glede na svetlobne pogoje, avtomatsko prižiganje in ugašanje luči, nadzor ogrevanja ali pa zalivanje vrta...

Celoten sistem je sestavljen iz več komponent, osredotočili pa se bomo na plošče, ki krmilijo porabnike na električnem omrežju. Poleg tega obstajajo še nadzorne enote z zaslonom na dotik, razni senzorji, v razvoju pa je tudi aplikacija za upravljanje preko interneta.

Vsaka plošča vsebuje mikrokrmilnik, 24 vhodov in 16 izhodov. Obstajajo tri vrste konfiguracij, ki se delijo glede na vrsto izhodov: 16 triak izhodov (t.i. T-plošča), 16 rele izhodov (t.i. R-plošča) in 8 rele ter 8 triak izhodov (t.i. RT-plošča). Izbira plošče zavisi od priključenega porabnika. Primer plošče T lahko vidimo na Sliki 2.4. Plošča vsebuje še druge elemente za delovanje, kot so dodatni pomnilniški elementi, optokoplerji, varovalke...

Plošče delujejo avtonomno. Na vsakega od 24 vhodov se priklopijo stenska stikala ali pa razni senzorji, kot so senzor gibanja ali pa senzor svetlobe. Na vsakem vhodu se razlikuje kratek oziroma dolg pritisk. Glede na vrsto dogodka na vhodu se izvrši akcija na enem ali več izhodih. Vrsta akcije je določena v nastavitvah, ki so shranjene v spominu EEPROM. Posamezna nastavitev vsebuje veliko parametrov, med drugim izhode, na katerih se naj izvrši funkcija, čas zakasnitve, čas delovanja pred izklopom...

Vsako ploščo lahko povežemo v omrežje preko vodila RS-485. Enote na ta način sprejemajo ukaze ter sporočajo spremembe stanja na izhodih, da lahko druge



Slika 2.4: T enota

naprave (LCD prikazovalnik, spletni strežnik) prikazujejo dejansko stanje. Na vodilo se lahko priklopimo tudi z osebnim računalnikom ter z namensko programsko opremo konfiguriramo sistem.

Poglavje 3

Prehod med platformama

V tem poglavju bomo opisali, s katerimi omejitvami smo se srečali pri razvoju sistema na trenutni platformi in zakaj bi bila potrebna zamenjava. Pogledali bomo, kaj nam obe platformi ponujata in pretehtali dobre in slabe strani migracije.

3.1 Omejitve pri razvoju na ATmega328P

V nadaljevanju so naštete težave, do katerih je priško z dodajanjem in razširjanjem funkcij pri razvoju sistema. Zaradi njih je nadgradnja sistema v prihodnosti nemogoča.

1. `Omejena zmogljivost.` Pri razvoju sistema smo se kar nekajkrat srečali z omejeno hitrostjo mikrokrmilnika. Določene rutine, ki so se izvajale periodično se preprosto niso dovolj hitro izvedle in do nove zahteve je prišlo, še preden se je stara izvedla do konca. Težave smo imeli tudi pri zatemnjevanju, saj se morajo določeni deli kode izvesti ob točno določenem času. Program smo morali zato dodatno optimizirati. Kot smo omenili v Poglavju 2, je v mikrokrmilnik vgrajen oscilator z maksimalno frekvenco 8 MHz. Če želimo izkoristiti polno hitrost, je potrebno uporabiti zunanji izvor ure, kar pa zaradi ponesrečene razporeditve pinov ni možno. Iz Slike 3.1 je razvidno, da signal ure pripeljemo na nožici 9 in 10. Opazimo tudi, da so na voljo le ena 8-bitna vrata - vrata B, ki se nahajajo na nožicah 14, 15, 16, 17, 18, 19, 9 in 10. Za prenašanje podatkov v sistemu potrebujemo celotna vrata B, s

tem pa izgubimo možnost zunanje ure. S hitrejšo frekvenco bi lahko CPU malo bolj zadihal, vendar to ni dolgoročna rešitev.

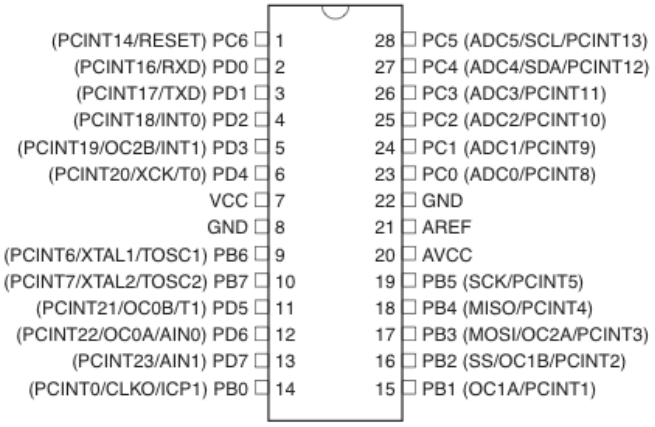
2. `Količina pomnilnika RAM.` Na voljo ga je 2K bajtov kar je zelo malo. Z razširjanjem sistema se povečuje tudi poraba pomnilnika RAM, saj vsaka funkcija potrebuje nekaj prostora. Zelo pomembno je, da pustimo delež prostega pomnilnika za delovanje sklada. Ker ATmega nima strojne zaščite pred prepisovanjem podatkov s strani sklada, je v programu potrebno preverjati, do kakšne globine seže. Pri razvoju sistema smo se posluževali metode, ki je opisana v Poglavju 4.4.

3. `Odsotnost prekinitvenega krmilnika.` Pri razvoju v takih sistemih pogosto uporabljamo prekinitve. Če jih imamo več, želimo imeti možnost prioritetnega razvrščanja. V Poglavju 4.2 bomo pojasnili, da mora imeti sinhronizacija absolutno prednost pred drugimi nalogami. ATmega ima sicer možnost vgnezdenih prekinitev. Pri tisti, za katero želimo, da ima najvišjo prioriteto, onemogočimo gnezdenje, pri vseh ostalih pa omogočimo, vendar to ni dovolj dobra rešitev.

4. `Pomanjkanje napredne periferije.` Vhodno izhodne naprave v mikrokrmilniku omogočajo komuniciranje z zunanjim svetom, generiranje signalov... Pri razvoju sistema bi nam prav prišlo predvsem več števcev, ki bi prevzeli breme generiranja signalov in tako razbremenili CPU. V Poglavju 4 je opisano, kako smo generiranje potrebnih signalov z uporabo strojne opreme minimizirali obremenitev CPU.

3.2 Prehod na arhitekturo ARM Cortex-M4

Mikrokrmilniki iz družine Cortex-M4 predstavljajo vrh ponudbe v razredu splošnonamenskih mikrokrmilnikov serije Cortex-M. Za primer prehoda na to družino smo se odločili, ker potrebujemo platformo, ki bo dovolj zmogljiva za vse nadgradnje sistema v prihodnosti. Pod pojmom zmogljivost imamo v mislih zmogljivost CPU kot tudi širok nabor vhodno-izhodnih naprav in vgrajenega spomina. Izbrani Cortex-M4 podjetja STM izpolnjuje vse zahteve iz Poglavja 3.1 in omogoča še



Slika 3.1: Razporeditev pinov na ATmega328P. Vir [1].

mnogo več. Omenimo nekaj najpomembnejših lastnosti mikrokrmilnika, ki igrajo pomembno vlogo pri izbiri:

1. `Hitrost CPU.` Centralna procesorska enota v mikrokrmilniku deluje do frekvence 168 MHz, kar omogoča visoko računsko zmogljivost.

2. `Bliskovni pomnilnik in pomnilnik RAM.` Vgrajenega je 1 MB bliskovnega pomnilnika, kar pomeni veliko prostora za program in konstantne podatke. 192 KB pomnilnika RAM omogoča uporabo programskih skladov, kot je TCP/IP, ki za svoje delovanje potrebujejo veliko prostora, vseeno pa bi ostalo kar nekaj pomnilnika za potrebe samega sistema. Mikrokrmilnik vsebuje tudi vmesnik za priklop zunanjega pomnilnika SRAM, kar omogoča še dodatno razširitev.

3. `Komunikacijski vmesniki.` Krmilnik podjetja STM vsebuje bogat nabor komunikacijskih vmesnikov. V sistemu se trenutno uporablja vodilo RS-485, ki pa je dovzeten za kolizije paketov. Smiselno bi bilo uporabiti katerega od vodil, ki so bolj prilagojena t.i. multi-master načinu, kot je CAN ali pa ethernet. Za uporabo etherneta bi sicer potrebovali še zunanji PHY vmesnik, saj je na čipu vsebovan le MAC krmilnik. Vgrajeni USB krmilnik bi lahko uporabili za nadgradnjo programske opreme.

4. `Števci.` Števci omogočajo merjenje časa, generiranje prekinitev in PWM signalov. V sistemu potrebujemo 16 neodvisnih PWM signalov, ki bi jih generirali s števci in tako razbremenili CPU. Tak način implementacije bomo prikazali v Poglavju 4.3.

5. `Dovolj nožic mikrokrmilnika.` Omenili smo že, da ima ATmega le ena 8-bitna vrata. Sistem vsebuje 16 izhodov ter 24 vhodov in za povezavo z njimi je potreben selektor ter dodatni pomnilniški elementi. Krmilnik na ploščici STM32F4 discovery ima 100 nožic, kar je dovolj za neposreden priklop vseh vhodov in izhodov, obstaja pa tudi 144-pinska verzija čipa, za katero bi se lahko odločili. Več nožic omogoča enostavnejše tiskano vezje in program.

6. `Druge periferne naprave.` Prav bi nam prišle tudi druge V/I naprave, kot so DMA krmilnik, kriptografski procesor in digitalno-analogni pretvornik.

Poraba energije sistema bi se sicer povečala. Jedri tako starega kot novega mikrokrmilnika imata sicer primerljivi porabi na MHz, vendar Cortex-M4 teče pri veliko višji frekvenci, poleg tega pa ima veliko več in kompleksnejših perifernih naprav, ki vplivajo na višjo porabo.

Prehod bi imel nekaj kratkoročnih dodatnih finančnih stroškov. Potrebno bi bilo ponovno zasnovati tiskano vezje, ki bi vsebovalo nov mikrokrmilnik. Nadomestiti bi bilo potrebno tudi preostale komponente in integrirana vezja, saj sistem trenutno deluje na 5V napetostnih nivojih. Dodatni vložek predstavlja tudi prilagoditev programske opreme za nov mikroprocesor. Večina kode je prenosljive, saj je sistem implementiran v programskem jeziku C, kar zahteva le zamenjavo prevajalnika. Ponovno bi bilo potrebno spisati le programsko plast, ki neposredno komunicira s strojno opremo (angl. hardware abstraction layer), ter del programa, ki skrbi za komunikacijo med mikrokrmilnikom in drugimi elementi na tiskanem vezju.

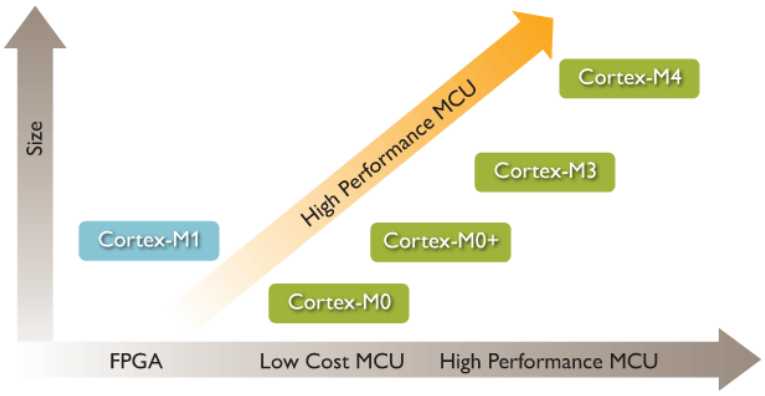
3.3 Primerjava z arhitekturami Cortex-M0 in Cortex-M3

Želimo, da je sistem čimbolj stroškovno optimiziran, zato je smiselno da pretehtamo, ali bi za nadgradnjo sistema zadostovala katera od manjših arhitektur kot

sta Cortex-M3 ali Cortex-M0.

Mikrokrmilniki, zasnovani okoli jedra Cortex-M0, predstavljajo vstopne modele mikrokrmilnikov iz družine Cortex-M. Imajo nizko ceno in majhno porabo, zato so neposredna konkurenca 8 in 16-bitnim mikrokrmilnikom, kot so Atmel AVR, TI MSP430, Microchip PIC, saj v podobnem cenovnem rangu ponujajo zmogljivost 32-bitne ARM arhitekture. Primerni so predvsem za sisteme, ki so baterijsko napajani. Za referenco smo vzeli mikrokrmilnike podjetja STM[15]. Omogočajo delovanje do frekvence 48 MHz, vsebujejo do 128k bajtov bliskovnega pomnilnika, do 16k bajtov pomnilnika SRAM ter druge V/I naprave, kot so serijski komunikacijski vmesniki, števci, USB in digitalno-analogni pretvorniki. Mikrokrmilniki tega tipa bi bila dobrodošla nadgradnja sistema v trenutnem obsegu. Vendar želimo platformo, ki bo v prihodnosti omogočala kompleksnejše nadgradnje. Na primer 16k bajtov pomnilnika SRAM je v primerjavi z 2k bajti v mikrokrmilniku ATmega ogromna nadgradnja, vendar na dolgi rok 16kB ni tako veliko. Ker je primarni cilj menjave platforme zmogljivost in ne manjša poraba energije, arhitektura Cortex-M0 po našem mnenju ni najboljša izbira.

Veliko bolj zanimiva izbira bi bila družina Cortex-M3. Družina teh mikrokrmilnikov je konceptualno podobna družini Cortex-M4. Razlika je v odsotnosti ukazov za DSP in enoti za računanje v plavajoči vejici, poleg tega tečejo pri manjši frekvenci in imajo manj pomnilnika RAM. Še vedno pa nudijo dobro zmogljivost ob nizki ceni. Potrebno bi bilo premisliti, ali posebne ukaze sploh potrebujemo in če programsko računanje v plavajoči vejici ne bi bistveno vplivalo na delovanje celotnega sistema. Sicer v mikrokrmilnikih Cortex-M3 najdemo enake oz. primerljive V/I naprave kot v Cortex-M4.



Slika 3.2: Družina Cortex-M. Vir [16].

Poglavje 4

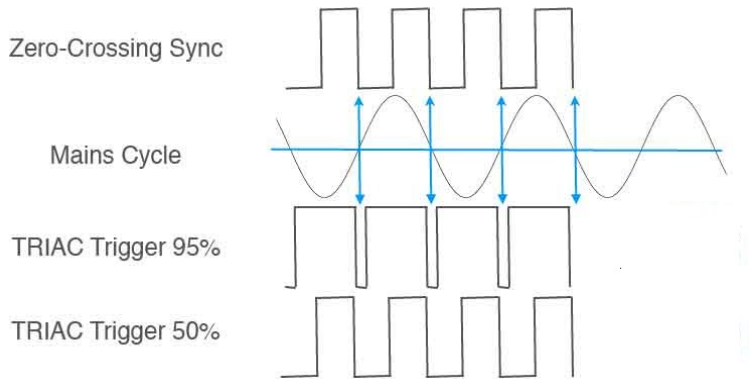
Zatemnjevanje

Zatemnjevanje (angl. dimming) je v sistemu ena od osnovnih funkcij. Uporablja se predvsem pri lučeh, s čimer v stanovanju prilagajamo svetilnost, prižigamo luči z naklonom (angl. fading) ter tako dosežemo prijeten ambient. V splošnem se lahko s to metodo nadzoruje na primer hitrost ventilatorja ali pa drugih naprav in hišnih pripomočkov na električnem omrežju.

4.1 Metoda zatemnjevanja

Zatemnjevanje dosežemo z uporabo elektronske komponente triak (angl. TRIAC). Ko je triak sprožen, prevaja električni tok v katerokoli smer, ob prehodu ničle v omrežni napetosti pa se ugasne in potrebno ga je ponovno sprožiti. Zmanjšanje napetosti dosežemo tako, da po prehodu ničle počakamo določen čas in nato vklopimo triak. Bolj kot se oddaljujemo od ničle, manjša napetost bo na izhodu. Na primer v Evropi omrežna napetost deluje s frekvenco 50Hz, kar pomeni da se vsak prehod ničle zgodi na približno 10 milisekund. Polovično efektivno napetost dosežemo z aktivacijo triaka po 5 milisekundah. Primer za 95% in 50% je podan na Sliki 4.1.

Če želimo krmiliti triake, moramo mikrokrmilnik sinhronizirati z električnim omrežjem. Za to skrbi signal, katerega generira posebno vezje za detekcijo ničle izmenične napetosti. Ta signal vežemo na vhod mikrokrmilnika, ki ima možnost generiranja prekinitve ali pa proženja periferne naprave. V nadaljevanju bomo predstavili implementacijo na ATmega ter na Cortex-M4.

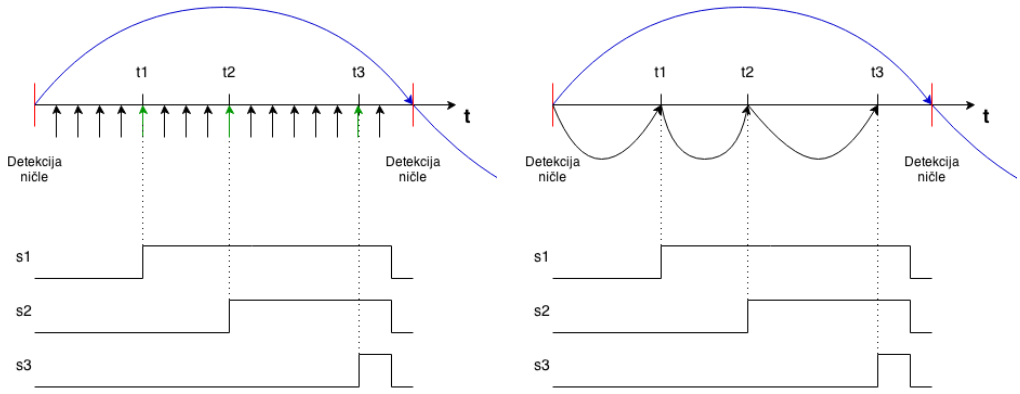


Slika 4.1: Proženje triaka. Vir [17].

4.2 Implementacija na ATmega328P

Naivna rešitev bi bila, da ob fiksnih točkah, ki so razmaknjene za t_{KORAK} preverimo, ali obstaja izhod, ki ga je potrebno aktivirati. Na primer ob resoluciji 128 korakov s kratkim izračunom ugotovimo, da je t_{KORAK}=78 mikrosekund. Torej bi bilo potrebno vsakih 78 mikrosekund iterirati čez vseh 16 izhodov ter preveriti, ali se časovna komponenta izhoda ujema s trenutno časovno točko. Iz Poglavja 3.1 vemo, da smo omejeni z uro 8 MHz. Taka metoda bi bila zelo neučinkovita, saj bi se procesor večinoma ukvarjal le s tem.

Poslužili smo se bolj inteligentne metode, pri kateri uporabimo števec in njegov primerjalni kanal (angl. output compare, v nadaljevanju OC). Vsakemu OC kanalu števca pripada register, čigar vrednost se vsako urino periodo števca primerja z glavnim števnim registrom. Če sta vrednosti enaki, se sproži prekinitvena zahteva. Ideja algoritma je v tem, da se prekinitve sprožijo le takrat, ko jih res potrebujemo, in tako učinkovito izrabljamo sistemske vire. To dosežemo s seznamom, v katerem hranimo pare *(odmik, izhodi)*, kjer je *odmik* časovni odmik od ničle, *izhodi* pa vektor izhodov, ki jih moramo aktivirati ob tem času. Seznam parov uredimo po časovni komponenti naraščujoče ter ob detekciji ničle zaženemo števec, v primerjalni register pa vpišemo prvo časovno točko, kot je prikazano na Sliki 4.2b. V prekinitveno servisnem programu števca aktiviramo izhode, ki pripadajo tej časovni točki, ter v primerjalni register vpišemo naslednjo časovno točko. To ponavljamo, dokler obstaja še kakšen izhod, ki ga je potrebno aktivirati.



(a) Naivni (b) Optimiziran

Slika 4.2: Ideji algoritma

V implementaciji smo uporabili 16-bitni števec TC1, ki bo tekel pri 1 MHz, ter zunanjo prekinitev INT1, na katero smo povezali signal detekcije ničle. TC1 smo konfigurirali v prostotekoči način ter omogočili prekinitev na output-compare kanalu A. Prekinitev se sproži ob enakosti registra COMPA z glavnim števnim registrom TCNT1.

Modul sestavljata dve funkciji, ki sta dostopni drugim delom kode. Njuna prototipa sta sledeča:

Koda 4.1: Prototipa funkcij

```
1 void dimmer_init(void);
2 void set_dimmer_value(uint8_t output_pin_index, uint8_t
    output_value);
```

Funkcija `dimmer_init` inicializira modul. Uporabimo jo na primer ob resetu. Zadolžena je za konfiguracijo V/I pinov, števca in prekinitev.
S funkcijo `set_dimmer_value` nastavimo vrednost svetilnosti `output_value` izhodu z indeksom `output_pin_index`.

Glede na vrednost svetilnosti ločimo tri primere:

1. Nastavljena svetilnost ima minimalno vrednost: signal nastavimo na nizko

stanje.

2. Nastavljena svetilnost ima maksimalno vrednost: signal nastavimo na visoko stanje.

3. Sicer sortiramo po vrednosti zatemnitve in uporabimo zgoraj opisan algoritem.

Opozoriti velja, da resolucija zatemnjevanja ni poljubna, saj je odvisna od zmogljivosti procesorja. Največje število korakov ocenimo tako, da preštejemo strojne ukaze prevedenega prekinitveno servisnega programa števca z metodo reverznega prevajanja (angl. de-compile). Čas koraka mora biti daljši od časa izvajanja PSP. Pri analizi Kode 4.2 smo ugotovili, da je varno uporabiti približno 150 korakov.

Koda 4.2: Modul za zatemnjevanje na ATmega328P

```
1  //Uvoz sistemskih knjiznic in definicij registrov
       mikrokontrolerja
2  #include <stdint.h>
3  #include <avr/io.h>
4  #include <avr/interrupt.h>
5
6  //Dostop do funkcij za delo z izhodi
7  #include "io.h"
8
9  //Definicije konstant
10 #define OUTPUTS 16 //Stevilo izhodov v sistemu
11 #define INPUT_MIN 0 //Najmnajsa mozna vrednost
12 #define INPUT_MAX 127 //Najvecja mozna vrednost
13 #define STEPS (INPUT_MAX-INPUT_MIN+1) //Stevilo korakov
14 #define STEP (10000/STEPS) //Sirina koraka
15
16 //Struktura, ki hrani vrednost zatemnitve, za izhode podane
       v vektorju
17 typedef struct{
18    uint16_t value;
19    uint16_t output_pins;
20 } dimmer_struct_t;
```

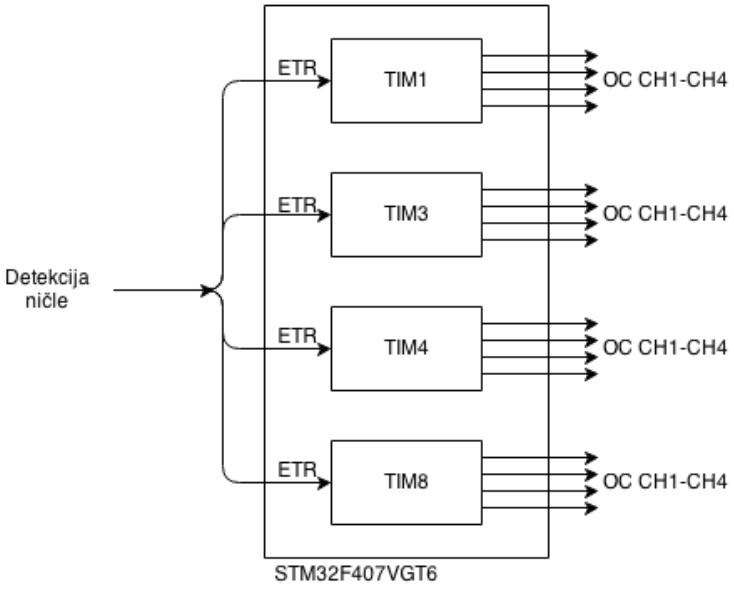
```
21
22 //Tabela, ki hrani zgornjo strukturo
23 static volatile dimmer_struct_t dimmer_values[OUTPUTS];
24 static volatile dimmer_struct_t dimmer_values_copy[OUTPUTS
       ];
25
26 //Lookup tabela, ki za vsako vrednost izhoda drzi 16bit
       vektor
27 //ki pove kateri izhodi se morajo prizgati
28 static uint16_t output_vectors[STEPS-1];
29
30 //Tabela, ki za vsak izhod drzi vrednost izhoda
31 static uint8_t output_values[OUTPUTS];
32
33 //Zastavica, za nove podatke
34 static volatile uint8_t new_data;
35
36 //Spremenljivka, ki hrani zadnji indeks
37 static volatile uint8_t end_index;
38 static volatile uint8_t end_index_copy;
39
40 //Trenutni index, kateri kaze na naslednjo urejeno vrednost
41 static volatile uint8_t dimmer_index;
42
43 //16-bitni vektor, ki za vsak izhod pove ali je dimmer
       omogocen
44 static volatile uint16_t dimmer_enabled;
45 static volatile uint16_t dimmer_enabled_copy;
46
47 /*
48 Funkcija za inicializacijo modula.
49 Potrebno je primerno konfigurirati vhod za detekcijo nicle,
       konfigurirati
50 stevec, ter omogociti potrebne prekinitve.
51 */
52 void dimmer_init(void){
53    //Nozico PD3 nastavi kot vhod
54    DDRD &= ~(1<<PD3);
```

```
55
56      //Omogoci zunanjo prekinitev INT1 - detekcija nicle
57      EICRA |= 1<<ISC10; //zaznavamo pozitivno in negativno
            fronto
58      EIMSK |= 1<<INT1; //omogocimo INT1
59
60      //Inicializacija stevca TC1; stevec pozene prekinitev ob
            detekciji nicle
61      //Nastavi prosto tekoci nacin
62      TCCR1A=0;
63      TCCR1B=0;
64      TCCR1C=0;
65      //omogoci prekinitev ob enakosti z registrom COMPA
66      TIMSK1 |= 1<<OCIE1A;
67  }
68
69  /*
70  Funkcija, ki nastavi vrednost zatemnjevanja output_value na
        indeksu izhoda output_pin_index
71   */
72  void set_dimmer_value(uint8_t output_pin_index, uint8_t
        output_value){
73      uint16_t mask=1<<output_pin_index;
74
75      //brisemo bit v tistem vektorju, kateri je bil
            postavljen v prejsnjem ciklu
76      output_vectors[output_values[output_pin_index]] &= ~mask
            ;
77
78      //zbrisemo zastavico o novem podatku
79      new_data=0;
80
81      //ce je vhod minimalni vrednosti vhoda onemogoci dimmer
            na tem pinu
82      //in ugasni triac
83      if(output_value == INPUT_MIN){
84          dimmer_enabled &= ~mask;
85          IO_clear_output_pin(output_pin_index);
```

```
86      }
87
88      //ce je vhod manjsi ali enak min vhodu onemogoci dimmer
            na tem pinu
89      //in prizgi triac
90      else if(output_value >= INPUT_MAX){
91          dimmer_enabled &= ~mask;
92          IO_set_output_pin(output_pin_index);
93      }
94
95      //vhod med min in max
96      else{
97          dimmer_enabled |= mask;
98
99          //v vektorju, ki ustreza novi vrednosti postavi bit
100         output_vectors[output_value] |= mask;
101         //zapisi novo vrednost
102         output_values[output_pin_index]=output_value;
103
104         uint8_t i,j;
105         j=0;
106         //skrcimo v tabelo dolgo OUTPUTS in inicializiramo
                struct-e
107         //i je iz intervala (INPUT_MIN, INPUT_MAX), saj robne
                primere obravnavamo zgoraj
108         for(i=INPUT_MAX-1; i>INPUT_MIN && j<OUTPUTS; i--){
109             //Ce je postavljen vsaj en bit
110             if(output_vectors[i] != 0){
111                 dimmer_values[j].output_pins=output_vectors[i];
112                 dimmer_values[j].value=(uint16_t)((INPUT_MAX-i)
                        *STEP);
113                 j++;
114             }
115         }
116         end_index=j;
117
118         //posatvimo zastavico, da so se podatki spremenili
119         new_data=1;
```

```
120     }
121 }
122
123 //Prekinitev ob detekciji nicle
124 ISR(INT1_vect){
125     //pozitivna fronta - zacetek nicle
126     if((PIND & (1<<PD3)) != 0){
127
128         //reset tistih triakov, nakaterih je omogocena
                sinhronizacija
129         IO_clear_output(dimmer_enabled_copy);
130         if(new_data){
131             uint8_t i;
132             //kopiramo vrednosti, da so podatki med niclama
                    stabilni
133             for(i=0; i<end_index; i++){
134                 dimmer_values_copy[i]=dimmer_values[i];
135             }
136             //brisemo zastavico za nove podatke
137             new_data=0;
138             //kopiramo indeks
139             end_index_copy=end_index;
140             //kopiramo masko
141             dimmer_enabled_copy=dimmer_enabled;
142         }
143         //postavimo indeks na zacetek
144         dimmer_index=0;
145
146         //preklicemo morebitno cakajoco prekinitev
147         TIFR1 = 1<<OCF1A;
148         //vrednost stevca postavimo na 0
149         TCNT1=0;
150     }
151
152     //negativna fronta - izhod iz nicle
153     else{
154         //ce je potrebno na katerem od izhodov izvesti
                zatemnjevanje,
```

```
155         //potem zazenemo stevec
156         if(end_index_copy > 0){
157             //v primerjalni register vpisemo prvo vrednost
158             OCR1A=(uint16_t)dimmer_values_copy[0].value;
159             //zazenemo stevec - omogocimo uro; uporabimo
                    delilnik frekvence s faktorjem 8
160             TCCR1B = 1<<CS11;
161         }
162     }
163 }
164
165 //prekinitev ob primerjavi
166 ISR(TIMER1_COMPA_vect){
167     //Vektor izhodov pri tem indeksu
168     uint16_t pins=dimmer_values_copy[dimmer_index].
            output_pins;
169     //postavimo omogocene izhode
170     IO_set_output(pins);
171
172     //povecamo indeks
173     dimmer_index++;
174
175     //Ce se nismo pri koncu v primerjalni register vpisi
176     //naslednjo vrednost za primerjanje
177     if(sync_index < end_index_copy){
178         OCR1A=dimmer_values_copy[dimmer_index].value;
179     }
180
181     //V nasprotnem primeru ustavimo stevec
182     else{
183         TCCR1B=0;
184     }
185 }
```



Slika 4.3: Uporaba števcev v mikrokrmilniku ARM Cortex-M4

4.3 Implementacija na ARM Cortex-M4

Kot smo omenili v Poglavju 2.3, ima mikrokrmilnik na razvojni ploščici STM32F4 discovery zelo bogat nabor števcev, kar smo v implementaciji izkoristili. V sistemu potrebujemo 16 neodvisnih signalov, zato smo uporabili štiri 16-bitne števce TIM1, TIM3, TIM4 in TIM8. Vsak ima štiri neodvisne OC kanale ter možnost proženja z zunanjim signalom. Uporabo števcev prikazuje Slika 4.3.

Osnova vsakega števca TIMx (kjer je x 1, 3, 4 oziroma 8) je register CNT, kateremu se ob vsaki pozitivni fronti ure spremeni vrednost za ena. Smer spremembe je odvisna od konfiguracije, v našem primeru uporabljamo štetje navzgor, zato bomo tako delovanje privzeli. Register CNT se povečuje, dokler ni dosežena enaka vrednost kot v registru ARR, ob enakosti pa se generira dogodek UEV (update event) in CNT se resetira na vrednost 0. Vsak OC kanal vsebuje svoj register CCyR (kjer je y 1, 2, 3 oziroma 4), kateri se vsako periodo ure primerja z registrom CNT, glede na rezultat primerjave pa se določi stanje na signalu CHy.

S števcem želimo ob negativni fronti na ETR vhodu generirati enkraten pulz z dolžino t_{PULZ}po zakasnitvi t_{ZAKAS}na vsakem OC kanalu, kar dosežemo z uporabo eno-pulznega načina delovanja (angl. one-pulse mode, OPM). V eno-pulznem

načinu se števec ob dogodku UEV samodejno deaktivira in za ponoven pulz je potreben nov dražljaj. Glede na nastavljeno svetilnost izberemo enega od načinov delovanja kanala CHy:

- Če je nastavljena svetilnost minimalna možna, prisilimo nizko stanje na CHy.
- Če je nastavljena svetilnost maksimalna možna, prisilimo visoko stanje na CHy.
- V ostalih primerih uporabimo način PWM2: signal CHy je nizek dokler je vrednost v registru CNT manjša od vrednosti v registru CCRy (t_{ZAKAS}), v nasprotnem primeru pa je aktiven (t_{PULZ}).

Resolucijo zatemnjevanja določimo z vrednostjo v registru ARR. Ob tem moramo seveda nastaviti dovolj hitro uro, da števec v eni polperiodi omrežne napetosti prešteje do vrednosti ARR. V implementaciji smo z delilnikom frekvence zmanjšali uro na 1 Mhz, kar pomeni 10000 urinih ciklov med ničlama. Posledično je s tem določena tudi vrednost v ARR registru. Opazimo, da povečevanje resolucije ne vpilva na obremenjenost CPU.

Nazadnje smo določili še, na katere nožice mikrokrmilnika zvežemo signale CHy in ETR. Postopek je sledeč:

- V navodilih za uporabo [14] pogledamo, na katere nožice so zvezani signali periferne naprave.
- Pogledamo, katere nožice so na voljo na mikrokrmilniku (v našem primeru na ploščici STM32F4 discovery).

Presek med zgornjima točkama nam da rešitev. Izbiro v našem primeru prikazuje Tabela 4.1.

Slabost te implementacije je poraba štirih nožic le za signal detekcije ničle. Implementacijo lahko prilagodimo tako, da uporabimo le eno nožico, ki proži prekinitev, v servisno-prekinitvene programu pa ročno poženemo števce. Spodaj je prikazana implementacija, ki ima enako strukturo kot v Poglavju 4.2.

Signal števca	**Nožica**	**Alternativna funkcija**
TIM1_ETR	PE7	1
TIM1_CH1	PE9	1
TIM1_CH2	PE11	1
TIM1_CH3	PE13	1
TIM1_CH4	PE14	1
TIM3_ETR	PD2	3
TIM3_CH1	PB4	2
TIM3_CH2	PB5	2
TIM3_CH3	PB0	2
TIM3_CH4	PB1	2
TIM4_ETR	PE0	2
TIM4_CH1	PB6	2
TIM4_CH2	PB7	2
TIM4_CH3	PB8	2
TIM4_CH4	PB9	2
TIM8_ETR	PA0	3
TIM8_CH1	PC6	3
TIM8_CH2	PC7	3
TIM8_CH3	PC8	3
TIM8_CH4	PC9	3

Tabela 4.1: Izbrane nožice mikrokrmilnika za signale števcev.

Koda 4.3: Modul za zatemnjevanje na STM32F4 Discovery[1]

```
#include <stdint.h>
#include "stm32f4xx.h"

#define INPUT_MIN 0
#define INPUT_MAX 10000

static const uint32_t TIM_base[]={
                TIM1_BASE, //TIM1 bazni naslov
                TIM3_BASE, //TIM3 bazni naslov
                TIM4_BASE, //TIM4 bazni naslov
                TIM8_BASE //TIM8 bazni naslov
                };

/*
Funkcija za inicializacijo modula.
Omogocimo uro za uporabljene V/I naprave, pinom
dolocimo ustrezne alternativne funkcije in
konfiguriramo stevce
*/
void dimmer_init(void){
   //Omogocimo uro za uporabljene naprave
   //Ura za TIM1 in TIM8
   RCC->APB2ENR |= RCC_APB2ENR_TIM1EN | RCC_APB2ENR_TIM8EN;
   //Ura za TIM3 in TIM4
   RCC->APB1ENR |= RCC_APB1ENR_TIM3EN | RCC_APB1ENR_TIM4EN;
   //Ura za GPIOA, GPIOB, GPIOC, GPIOD, GPIOE
   RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN |
       RCC_AHB1ENR_GPIOBEN
                | RCC_AHB1ENR_GPIOCEN | RCC_AHB1ENR_GPIODEN
                | RCC_AHB1ENR_GPIOEEN;

   //Inicializacija V/I pinov
   //PA0: TIM8 ETR
   GPIOA->MODER |= 0x2; //Alternativna funkcija pina
   GPIOA->AFR[0] |= 0x3; //AF3
```

```
35
36      //PB4, PB5, PB0, PB1: TIM3 CH1-CH4, PB6, PB7, PB8, PB9:
            TIM4 CH1-CH4
37      GPIOB->MODER |= 0x2 //PB0
38                   | 0x2 << 2 //PB1
39                   | 0x2 << 8 //PB4
40                   | 0x2 << 10 //PB5
41                   | 0x2 << 12 //PB6
42                   | 0x2 << 14 //PB7
43                   | 0x2 << 16 //PB8
44                   | 0x2 << 18; //PB9
45
46      //AF2 za PB0, PB1, PB4, PB5, PB6, PB7
47      GPIOB->AFR[0] |= 0x2 //PB0
48                    | 0x2 << 4 //PB1
49                    | 0x2 << 16 //PB4
50                    | 0x2 << 20 //PB5
51                    | 0x2 << 24 //PB6
52                    | 0x2 << 28; //PB7
53      //AF2 za PB8, PB9
54      GPIOB->AFR[1] |= 0x2 //PB8
55                    | 0x2 << 4; //PB9
56
57      //PC6, PC7, PC8, PC9: TIM8 CH1-CH4
58      GPIOC->MODER |= 0x2 << 12 //PC6
59                   | 0x2 << 14 //PC7
60                   | 0x2 << 16 //PC8
61                   | 0x2 << 18; //PC9
62
63      //AF3 za PC6, PC7
64      GPIOC->AFR[0] |= 0x3 << 24 //PC6
65                    | 0x3 << 28; //PC7
66
67      //AF3 za PC8, PC9
68      GPIOC->AFR[1] |= 0x3 //PC8
69                    | 0x3 << 4; //PC9
70
71      //PD2: TIM3 ETR
```

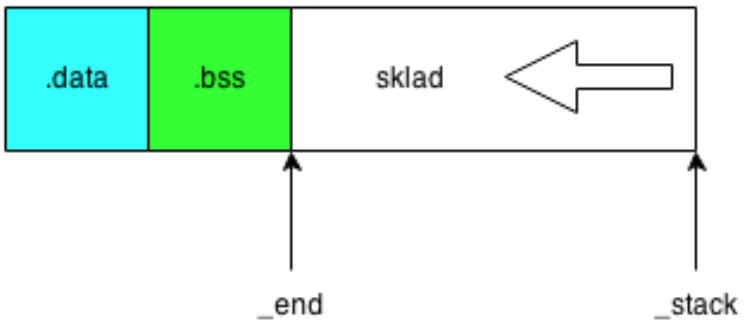
```
72      GPIOD->MODER |= 0x2 << 4;
73
74      //AF3 za PD2
75      GPIOD->AFR[0] |= 0x3 << 8;
76
77      //PE0: TIM4 ETR; PE7: TIM1 ETR; PE9, PE11, PE13, PE14:
            TIM1 CH1-CH4
78      GPIOE->MODER |= 0x2 //PE0
79                   | 0x2 << 14 //PE7
80                   | 0x2 << 18 //PE9
81                   | 0x2 << 22 //PE11
82                   | 0x2 << 26 //PE13
83                   | 0x2 << 28; //PE14
84
85      //AF2 za PE0, AF1 za PE7
86      GPIOE->AFR[0] |= 0x2 //PE0
87                   | 0x1 << 28; //PE7
88
89      //AF1 za PE9, PE11, PE13, PE14
90      GPIOE->AFR[1] |= 0x1 << 4 //PE9
91                   | 0x1 << 12 //PE11
92                   | 0X1 << 20 //PE13
93                   | 0x1 << 24; //PE14
94
95
96      //Inicializacija timerjev
97      uint32_t i;
98      TIM_TypeDef * tim;
99      for(i=0; i<4; i++){
100        tim=(TIM_TypeDef *)TIM_base[i];
101
102        //One-pulse nacin
103        tim->CR1=1<<3;
104
105        tim->SMCR = 1 << 15 //ETP: ETR aktiven ob negativni
               fronti
106                | 0x7 << 4 // TS: External trigger
107                | 0x6; //SMS: Trigger mode
```

```
108
109         uint16_t ccmr = 0x04 << 4 //force inactiove level
110                       | 1 << 3; //OC preload enable
111
112         //enako tudi za drugo polovico registra
113         ccmr |= ccmr << 8;
114         tim->CCMR1 = ccmr;
115         tim->CCMR2 = ccmr;
116
117         //stej do 10000
118         tim->ARR=10000;
119     }
120
121     //S prescaler-jem zmanjsamo uro na 1 MHz
122     //TIM3 in TIM4 poganja 84 MHz ura
123     TIM3->PSC=84;
124     TIM4->PSC=84;
125     //TIM1 in TIM8 poganja 168 MHz ura
126     TIM1->PSC=168;
127     TIM8->PSC=168;
128
129     //Sprozi UEV in pocakaj na zunanji stimulus
130     tim->EGR = 1;
131 }
132
133 /*
134 Funkcija, ki nastavi vrednost zatemnjevanja output_value na
        indeksu izhoda output_pin_index
135  */
136 void set_dimmer_value(uint8_t output_pin_index, uint16_t
        output_value){
137     uint32_t tim=TIM_base[output_pin_index>>2];
138
139     //izracunamo ali pademo v register TIMx_CCMR1 ali
            TIMx_CCMR2
140     //TIMx_CCMR1 ima odmik 0x18, ce pa je potrebna uporaba
            TIMx_CCMR2
141     //pristejemo 4 (TIMx_CCMR2 ima odmik 0x1C)
```

```
142     uint32_t ccmr_offset=(uint32_t)(0x18 + ((
            output_pin_index<<1) & 0x04));
143
144     //kazalec na register TIMx_CCMR1 / TIMx_CCMR1
145     volatile uint16_t * ccmr_p=(volatile uint16_t *)(tim+
            ccmr_offset);
146
147     //preberi trenutno vsebino registra
148     uint16_t ccmr=*ccmr_p;
149
150     //izracunamo v katero polovico registra pademo
151     uint32_t shift=(uint32_t)((output_pin_index & 1) << 3);
152
153     //brisemo bite
154     ccmr &= (uint16_t)~(TIM_CCMR1_OC1M << shift);
155
156     //robni pogoj
157     if(output_value == INPUT_MIN){
158        //prisilimo nizko stanje: 0b100 v OC1M / OC2M
159        ccmr |= (uint16_t)((0x0004 << 4) << shift);
160     }
161
162     //robni pogoj
163     else if(output_value >= INPUT_MAX){
164        //prisilimo visoko stanje: 0b101 v OC1M / OC2M
165        ccmr |= (uint16_t)((0x0005 << 4) << shift);
166     }
167     else{
168        //PWM2 mode: 0b111
169        ccmr |= (uint16_t)((0x0007 << 4) << shift);
170        //Izracunamo kazalec na CCR register
171        volatile uint32_t* ccr;
172        ccr=((volatile uint32_t *)(tim + 0x34)) + (
               output_pin_index & 0x03);
173        //Zapisemo v CCR register
174        *ccr=(uint32_t)(INPUT_MAX - output_value);
175     }
176     //zapisi CCMR
```

```
177     *ccmr_p=ccmr;
178 }
```

4.4 Algoritem za preverjanje globine sklada



Slika 4.4: RAM

Ob implementaciji sistema na mikrokrmilniku ATmega smo se srečali s skrajno porabo pomnilnika RAM. Zavedati se moramo, da pomnilnik poleg statičnih podatkov uporablja tudi dinamična struktura sklad, ki se uporablja za shranjevanje registrov ob klicih podprogramov, prenos parametrov in shranjevanje lokalnih spremenljivk. V času prevajanja ni mogoče natančno določiti porabo pomnilnika s strani sklada, zato v ta namen uporabimo algoritem, ki bo deloval med izvajanjem samega programa, ki ga opazujemo.

Slika 4.4 prikazuje tipično stanje RAM-a po inicializaciji C okolja. Kadar je na voljo RAM-a malo ali pa uporabljamo preprost mikrokrmilnik brez strojne zaščite pomnilnika, kot je ATmega328P, lahko pride do tega, da sklad začne prepisovati statične podatke - .bss in .data sekcijo (odvisno od umestitve V/I naprav v spominski prostor pa tudi njihove registre). To lahko detektiramo s spodnjim algoritmom.

1. Pred zagonom aplikacije "poslikamo"RAM med podatki in začetkom sklada z neko konstantno vrednostjo.

[1]Koda zaradi poenastavitve predpostavlja stanje registrov ob resetu

2. Ko želimo preveriti koliko prostora ima sklad še na voljo, pokličemo funkcijo, ki prešteje bajte z vrednostjo iz točke 1.

Koda 4.4 je implementirana za uporabo s C knjižnico avr-libc in s prevajalnikom avr-gcc.

Koda 4.4: Algoritem za preverjanje globine sklada

```
1  #include <stdint.h>
2
3  //pridobimo dostop do spremnljivk linkerja
4  extern uint8_t _end; //prvi prost bajt po .bss sekciji
5  extern uint8_t __stack; //naslov zacetka sklada
6
7  //definiramo vrednost kanarcka
8  #define CANARY 0xC5;
9
10 //funkcija stack_paint zelimo da se izvede v sekciji init1
11 void stack_paint(void) __attribute__ ((naked))
       __attribute__ ((section (".init1")));
12
13 //Napolnimo RAM od .bss sekcije do zacetka sklada
14 void stack_paint(void){
15    uint8_t * p=&_end;
16    while(p <= &__stack){
17       *p=CANARY;
18       p++;
19    }
20 }
21
22 //Funkcija vrne stevilo bajtov od .bss sekcije do prvrga
       bajta, ki nima vrednosti kanarcka,
23 //kar nam pove koliko bajtov ima sklad se rezerve do .bss
       podatkov
24 //ce vrne 0 smo v tezavah
25 uint16_t stack_count(void){
26    uint8_t * p=&_end;
27    uint16_t count=0;
```

```
28      while(*p == CANARY && p <= &__stack){
29          p++;
30          count++;
31      }
32
33      return count;
34  }
```

Poglavje 5

Sklepne ugotovitve

S prehodom na arhitekturo ARM Cortex-M4 bi sistem predvsem lažje zadihal, saj smo na trenutni arhitekturi dosegli zmogljivostno mejo. Največji oviri predstavljata predvsem pomankanje pomnilnika RAM ter hitrost procesorja, želeli pa bi tudi več vhodno-izhodnih naprav. Posledično so onemogočene nadaljnje izboljšave in nadgradnje sistema, kar predstavlja zaostanek na izredno konkurenčnem področju.

Z zmogljivejšim mikrokrmilnikom, bi lahko več dela prepustili strojni opremi kar bi poenostavilo razvoj ter pripomoglo k stabilnosti sistema. Ker smo sistem na mikrokrmilniku ATmega implementirali v programskem jeziku C, bi bilo potrebno spremeniti le strojno-abstrakcijski nivo, saj ciljna platforma prav tako omogoča pisanje programov v tem jeziku. Potrebna je le zamenjava prevajalnika.

Arhitektura ARM Cortex-M4 je v primerjavi z mikrokrmilniki ATmega kompleksnejša, zahteva več branja dokumentacije in posledično daljši čas razvoja. Podjetje STM nudi brezplačna orodja in knjižnice, ki zmanjšujejo ta čas in olajšajo delo programerju. Velik strošek prehoda predstavlja prilagoditev oziroma ponovna zasnova tiskanega vezja. Potrebno bi bilo izbrati nove komponente, saj trenutna logika deluje na višjem napetostnem nivoju. Ker pa ima nov mikrokrmilnik več nožic, bi lahko število uporabljenih komponent celo zmanjšali, saj dodatni pomnilniški elementi ne bi bili več potrebni. S tem bi kompenzirali nekoliko višjo ceno mikrokrmilnika ARM v primerjavi z ATmega. S prehodom na večji mikrokrmilnik bi se povečala tudi poraba energije, vendar je s pridobljeno zmogljivostjo ta lastnost upravičena.

Literatura

[1] Atmel ATmega328P. Dostopno na: http://www.atmel.com/Images/doc8161.pdf

[2] Dostopno na: http://img.banggood.com/thumb/view/upload/2012/lidanpo/SKU121232a.JPG

[3] Dostopno na: http://letsmakerobots.com/files/field_primary_image/usbaspver2.jpg

[4] Razvojno okolje Eclipse. Dostopno na: http://www.eclipse.org/

[5] AVR Eclipse vtičnik za razvojno okolje Eclipse. Dostopno na: http://avr-eclipse.sourceforge.net/wiki/index.php/The_AVR_Eclipse_Plugin

[6] GNU ARM Eclipse vtičnik za razvojno okolje Eclipse. Dostopno na: http://sourceforge.net/projects/gnuarmeclipse/

[7] USBasp programator za družino mikrokrmilnikov AVR. Dostopno na: http://www.fischl.de/usbasp/

[8] Razvojna ploščica STM32F4 discovery. Dostopno na: http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/PF252419

[9] STM32F40xx referenčna navodila RM0090. Dostopno na: http://www.st.com/web/en/resource/technical/document/reference_manual/DM00031020.pdf

[10] Dostopno na: http://2.bp.blogspot.com/-M5cd_nF-WKw/UECw8h10NSI/AAAAAAAAAM4/z12n5_I6fCQ/s1600/stm32f4_discovery.jpg

[11] Prevajalnik za družino AVR. Dostopno na: http://sourceforge.net/projects/winavr/files/

[12] Prevajalnik za ARM Cortex-M. Dostopno na: https://launchpad.net/gcc-arm-embedded

[13] Assys, sistem za avtomatizacijo hiš. Dostopno na: http://www.assys.si/

[14] Tabela 9: Alternate function mapping, strani 60-68 STM32F405xx/STM32F407xx datasheet. Dostopno na: http://www.st.com/web/en/resource/technical/document/datasheet/DM00037051.pdf

[15] STM32 32-bit ARM Cortex MCU selector. Dostopno na: http://www.st.com/web/en/catalog/mmc/FM141/SC1169

[16] Dostopno na: http://www.arm.com/images/roadmap/Cortex-M_Roadmap.gif

[17] Dostopno na: http://www.erg.abdn.ac.uk/~gorry/eg3576/Images/Triac-waveform-diag.jpg